

Programación Funcional Avanzada

Transformadores de Monads

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad “Simón Bolívar”

Copyright ©2010-2013

Caracterización de cómputo

Monad define el proceso abstracto

- Un Monad modela una característica particular del cómputo.
- El lenguaje ya establece varios modelos generales.
 - Maybe – Cómputo con resultado o falla genérica.
 - List – Cómputo con múltiples resultados.
 - Error – Cómputo con resultado o descripción de falla.
 - Reader – Ambiente de referencia “sólo lectura”.
 - Writer – Colector que registra valores.
 - State – Estado mutable.
- Monad “a la medida” para comportamientos específicos – Parsec y otros por estudiar. . .



Caracterización de cómputo

Monad define el proceso abstracto

- Un Monad modela una característica particular del cómputo.
- El lenguaje ya establece varios modelos generales.
 - `Maybe` – Cómputo con resultado o falla genérica.
 - `List` – Cómputo con múltiples resultados.
 - `Error` – Cómputo con resultado o descripción de falla.
 - `Reader` – Ambiente de referencia “sólo lectura”.
 - `Writer` – Colector que registra valores.
 - `State` – Estado mutable.
- Monad “a la medida” para comportamientos específicos – `Parsec` y otros por estudiar. . .

¿Y si queremos **combinar** comportamientos?



La clase MonadPlus de Control.Monad

Preludio a la composición de cálculos

- Cálculos monádicos para combinar *alternativas* o expresar *fracaso*.
- Maybe y List – casos particulares.
 - Ambos modelan fracaso, aunque de diferentes formas.
 - ¿Cómo combinar dos o más cálculos y escoger los resultados?
- MonadPlus – cualquier Monad para el cual sea posible definir

```
class Monad m => MonadPlus m where
  mzero  :: m a
  mplus  :: m a -> m a -> m a
```

- mzero – Fracaso.
- mplus – Combinar éxitos, de ser posible.



Instancias de MonadPlus

- `[a]` es instancia de `MonadPlus`

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```



Instancias de MonadPlus

- `[a]` es instancia de `MonadPlus`

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

- `Maybe a` es instancia de `MonadPlus` – sesgada al primer resultado.

```
instance MonadPlus Maybe where
  mzero = Nothing
  Nothing 'mplus' Nothing = Nothing
  Just x 'mplus' Nothing = Just x
  Nothing 'mplus' Just y = Just y
  Just x 'mplus' Just y = Just x    -- Ojo
```

Instancias de MonadPlus

El Monad para errores – Error

- Monad Error – Either e a es similar a Maybe a
 - Right a – resultado exitoso de tipo a.
 - Left e – fracaso con “explicación” de tipo e.
- Either e es instancia de MonadPlus – sesgada como Maybe

```
instance (Error e) => MonadPlus (Either e) where
  mzero           = Left ""
  Left _ 'mplus' n = n
  Right x 'mplus' _ = Right x
```

Queremos generalizar todavía más...



Transformadores

¿Y si queremos combinar comportamientos de forma general?

- Un **transformador** es un constructor de tipos:
 - Opera sobre un Monad subyacente.
 - Agrega un comportamiento monádico específico.
 - Produce un Monad “aumentado” con **ambos** comportamientos.



Transformadores

¿Y si queremos combinar comportamientos de forma general?

- Un **transformador** es un constructor de tipos:
 - Opera sobre un Monad subyacente.
 - Agrega un comportamiento monádico específico.
 - Produce un Monad “aumentado” con **ambos** comportamientos.
- Cada Monad en el lenguaje dispone de un **transformador**.
 - En forma de librería estándar.
 - Las combinaciones quedan a juicio (y práctica) del programador.
 - El orden de combinación no siempre es obvio.



Transformadores

¿Y si queremos combinar comportamientos de forma general?

- Un **transformador** es un constructor de tipos:
 - Opera sobre un Monad subyacente.
 - Agrega un comportamiento monádico específico.
 - Produce un Monad “aumentado” con **ambos** comportamientos.
- Cada Monad en el lenguaje dispone de un **transformador**.
 - En forma de librería estándar.
 - Las combinaciones quedan a juicio (y práctica) del programador.
 - El orden de combinación no siempre es obvio.
- Para obtener un Monad a la medida
 - Se escoge un Monad *base* – generalmente Identity, [] o IO.
 - Se le aplica un transformador que agrega funcionalidad “encima”.
 - Enguajar y repetir.



¿Qué es un Monad Transformer?

Salgamos de esto

- `Control.Monad.Trans` – transformadores puros.

```
class MonadTrans t where  
  lift :: (Monad m) => m a -> t m a
```

- El transformador `t` “envuelve” al `Monad m`.
- `lift` – sube el cómputo del `Monad` envuelto hasta envoltorio.



¿Qué es un Monad Transformer?

Salgamos de esto

- `Control.Monad.Trans` – transformadores puros.

```
class MonadTrans t where  
  lift :: (Monad m) => m a -> t m a
```

- El transformador `t` “envuelve” al `Monad m`.
 - `lift` – sube el cómputo del `Monad` envuelto hasta envoltorio.
- `Control.Monad.IO` – transformadores *sobre* `IO`.

```
class (Monad m) => MonadIO m where  
  liftIO :: IO a -> m a
```

- `liftIO` – para los transformadores que envuelven a `IO`
 - Esto **no** es escaparse del `Monad IO`.

Transformadores estándar

Monad	Tipo	Transformador	Tipo Final
Error	<code>Either e a</code>	<code>ErrorT</code>	<code>m (Either e a)</code>
State	<code>s -> (a,s)</code>	<code>StateT</code>	<code>s -> m (a,s)</code>
Reader	<code>r -> a</code>	<code>ReaderT</code>	<code>r -> m a</code>
Writer	<code>(a,w)</code>	<code>WriterT</code>	<code>m (a,w)</code>
List	<code>[a]</code>	<code>ListT</code>	<code>m [a]</code>

- Cada uno cuenta con una función para iniciar el cómputo combinado – `runErrorT`, `evalStateT`, `execWriterT`, ...
- `MaybeT` fue incorporado hace poco a las librerías – será nuestro ejemplo de estudio inicial.

Construyendo MaybeT

Al principio, está el tipo

- El Monad Maybe originario

```
data Maybe a = Nothing | Just a
```

modela la posibilidad de un cómputo con un resultado y posible falla.



Construyendo MaybeT

Al principio, está el tipo

- El Monad Maybe originario

```
data Maybe a = Nothing | Just a
```

modela la posibilidad de un cómputo con un resultado y posible falla.

- El transformador necesita

```
newtype MaybeT m a =  
    MaybeT { runMaybeT :: m (Maybe a) }
```

- Modificar un cómputo `m` envolviendo su resultado con un `Maybe`.
- `runMaybeT` – iniciar el cómputo transformado desenvolviendo al final.



Construyendo MaybeT

Luego, la operación interna

MaybeT produce un Monad, por lo tanto

```
instance (Monad m) => Monad (MaybeT m) where
  return a = MaybeT $ return (Just a)
  x >>= f  = MaybeT $ do c <- runMaybeT x
                    case c of
                      Nothing -> return Nothing
                      Just v   -> runMaybeT $ f v
```

- return usa el return subyacente.
- >>= opera en el monad subyacente (por eso el do).



¿Cómo está construido MaybeT?

Finalmente, levantar al subyacente

MaybeT es un MonadTrans, por lo tanto

```
instance MonadTrans MaybeT where  
  lift c = MaybeT $ Just 'liftM' c
```

- c es un cómputo en el Monad subyacente.
- MaybeT debe inyectar Just sobre el resultado del cómputo.



¿Cómo está construido MaybeT?

Ya que estamos ...

MaybeT podría generar un MonadPlus, por lo tanto

```
instance (Monad m) => MonadPlus (MaybeT m) where
  mzero          = MaybeT $ return Nothing
  x 'mplus' y = MaybeT $
    do c <- runMaybeT x
       case c of
         Nothing    -> runMaybeT y
         Just value -> runMaybeT x
```

- El fracaso se indica inyectando Nothing en el Monad subyacente.
- La combinación de resultados ocurre como en Maybe, prefiriendo el primero – noten que es *necesario* desenvolver y envolver.



Un ejemplo simple ...

Solicitar la nueva clave y determinar si es buena – *ad nauseam*

```
askPassword :: MaybeT IO ()
askPassword = do value <- msum $ repeat getPassword
                lift $ putStrLn "Changing..."

isGood s = length s >= 8 &&
          any isAlpha s && any isNumber s

getPassword :: MaybeT IO String
getPassword = do lift $ putStrLn "New password:"
                s <- lift getLine
                guard (isGood s) -- mzero :)
                return s

> runMaybeT askPassword
New password: malo
New password: G4bpq3yl!
Changing...
```

LIVAR

Otro ejemplo simple ...

Ahora usando StateT sobre IO

- Incrementar un contador y mostrarlo.
 - Un Monad State para guardar el estado ...
 - ...envolviendo el Monad IO.

```
tickAndPrint :: StateT Int IO ()
tickAndPrint = modify (+1) >> get >=> lift . print

> runStateT (sequence_ $ replicate 3 tickAndPrint)
39
40      -- Emitido por print
41      -- Emitido por print
42      -- Emitido por print
((),42)  -- Resultado de runStateT
```

- modify y get operan en el State “superior”.
- print opera en el IO “inferior” así que debemos “levantarlo”.



¿Cómo está construido StateT?

Al principio, está el tipo

El Monad State originario

```
newtype State s a =  
    State { runState :: (s -> (a,s)) }
```

El transformador necesita

```
newtype StateT s m a =  
    StateT { runStateT :: (s -> m (a,s)) }
```



¿Cómo está construido StateT?

Luego, la operación interna

StateT produce un Monad, por lo tanto

```
instance (Monad m) => Monad (StateT s m) where
  return a           = StateT $ \s -> return (a,s)
  (StateT x) >>= f    =
    StateT $ \s -> do (v,s') <- x s
                      (StateT x') <- return $ f v
                      x' s'
```

- return debe usar el return subyacente.
- >>= necesita una acción (do) para actuar en el monad subyacente.



¿Cómo está construido StateT?

Luego, la operación interna

StateT produce un MonadState, por lo tanto

```
instance (Monad m) => MonadState (StateT s m) where
  get    = StateT $ \s -> return (s,s)
  put s = StateT $ \_ -> return ((),s)
```

- En ambos casos hay que usar return subyacente.



¿Cómo está construido StateT?

Luego, la operación interna

StateT produce un MonadState, por lo tanto

```
instance (Monad m) => MonadState (StateT s m) where
  get    = StateT $ \s -> return (s,s)
  put s = StateT $ \_ -> return ((),s)
```

- En ambos casos hay que usar return subyacente.

StateT puede producir un MonadPlus, por lo tanto

```
instance (MonadPlus m) => Monadplus (StateT s m) where
  mzero = StateT $ \s -> mzero
  (StateT x1) 'mplus' (StateT x2) =
    StateT $ \s -> (x1 s) 'mplus' (x2 s)
```

- Hay que usar mzero y mplus subyacentes.

¿Cómo está construido StateT?

Finalmente, levantar al subyacente

StateT es un MonadTrans, por lo tanto

```
instance MonadTrans (StateT s) where
  lift c = StateT $
    \s -> c >>= (\x -> return (x,s))
```

- c es un cómputo en el Monad subyacente.
- StateT debe producir un transformador de estado – por eso la función que opera sobre s .
- El $>>=$ y `return` están operando en el Monad subyacente.



Un ejemplo complicado

Las ocho reinas – Jeff Newbern

- Problema de las ocho reinas resuelto por “fuerza bruta”.
- Combinación de Monads
 - `[]` para cómputo no-determinístico – explorar el espacio de soluciones.
 - `StateT` para preservar el estado.
 - `WriterT` para registrar los pasos del evaluador.



Un ejemplo complicado

Las ocho reinas – Jeff Newbern

- Problema de las ocho reinas resuelto por “fuerza bruta”.
- Combinación de Monads
 - `[]` para cómputo no-determinístico – explorar el espacio de soluciones.
 - `StateT` para preservar el estado.
 - `WriterT` para registrar los pasos del evaluador.
- ¿Cuál combinación usar?



Un ejemplo complicado

Las ocho reinas – Jeff Newbern

- Problema de las ocho reinas resuelto por “fuerza bruta”.
- Combinación de Monads
 - `[]` para cómputo no-determinístico – explorar el espacio de soluciones.
 - `StateT` para preservar el estado.
 - `WriterT` para registrar los pasos del evaluador.
- ¿Cuál combinación usar?
 - `StateT s (WriterT w [])` con tipo `s -> [((a,s),w)]`



Un ejemplo complicado

Las ocho reinas – Jeff Newbern

- Problema de las ocho reinas resuelto por “fuerza bruta”.
- Combinación de Monads
 - `[]` para cómputo no-determinístico – explorar el espacio de soluciones.
 - `StateT` para preservar el estado.
 - `WriterT` para registrar los pasos del evaluador.
- ¿Cuál combinación usar?
 - `StateT s (WriterT w [])` con tipo `s -> [((a,s),w)]`
 - `WriterT w (StateT s [])` con tipo `s -> [((a,w),s)]`



Un ejemplo complicado

Las ocho reinas – Jeff Newbern

- Problema de las ocho reinas resuelto por “fuerza bruta”.
- Combinación de Monads
 - `[]` para cómputo no-determinístico – explorar el espacio de soluciones.
 - `StateT` para preservar el estado.
 - `WriterT` para registrar los pasos del evaluador.
- ¿Cuál combinación usar?
 - `StateT s (WriterT w [])` con tipo `s -> [((a,s),w)]`
 - `WriterT w (StateT s [])` con tipo `s -> [((a,w),s)]`
 - Da lo mismo – el autor decidió usar el segundo.
- Sólo discutiremos las partes relevantes del código – el resto deben estudiarlo del programa completo.



Modelo del estado y estado inicial

```
data NQueensProblem = NQP { board :: Board,
                             ranks  :: [Rank],
                             files  :: [File],
                             asc    :: [Diagonal],
                             desc   :: [Diagonal]
                           }

initialState =
  let fileA = map (\r->Pos A r) [1..8]
      rank8  = map (\f->Pos f 8) [A .. H]
      rank1  = map (\f->Pos f 1) [A .. H]
      asc    = map Ascending (nub (fileA ++ rank1))
      desc   = map Descending (nub (fileA ++ rank8))
  in NQP (Board []) [1..8] [A .. H] asc desc
```



Combinación de Monads

Non-Deterministic State

- Se usa un *type alias* para simplificar las firmas.

```
type NDS a =  
    WriterT [String] (StateT NQueensProblem []) a
```

- Para obtener una solución al problema ...
 - Iniciar el cómputo con el estado inicial.
 - Obtener el resultado en el Monad List – si existe vendrá acompañado de la bitácora de operaciones.

```
getSolution :: NDS a -> NQueensProblem  
            -> Maybe (a,[String])  
getSolution c i = listToMaybe $  
                    evalStateT (runWriterT c) i
```



Aquí es donde ocurre la magia

Poner una reina en la posición p

```
addQueen :: Position -> NDS ()
addQueen p = do (Board b) <- gets board
                rs <- gets ranks
                fs <- gets files
                as <- gets asc
                ds <- gets desc
                let b' = (Piece Black Queen, p):b
                    rs' = delete (rank p) rs
                    fs' = delete (file p) fs
                    (a,d) = getDiags p
                    as' = delete a as
                    ds' = delete d ds
                tell ["Added Queen at " ++ (show p)]
                put (NQP (Board b') rs' fs' as' ds')
```



Aquí es donde ocurre la magia

Poner reinas en todos los lugares viables

```
addQueens :: NDS ()
addQueens = do
  rs <- gets ranks
  fs <- gets files
  allowed <- filterM inDiags
    [Pos f r | f <- fs, r <- rs]
  tell [show (length allowed) ++ " possible choices"]
  msum (map addQueen allowed)

inDiags :: Position -> NDS Bool
inDiags p = do let (a,d) = getDiags p
  as <- gets asc
  ds <- gets desc
  return $ (elem a as) && (elem d ds)
```



El programa principal es directo

```
main :: IO ()
main = do
  args <- getArgs
  let n    = read (args!!0)
      cmds = replicate n addQueens
      sol  = ('getSolution' initialState) $ do
        sequence_ cmds
        gets board
  case sol of
    Just (b,l) -> do putStr $ show b
                     putStr $ unlines l
    Nothing     -> putStrLn "No solution"
```

- ¿Cuántas reinas? – argumento en línea de comandos.
- Agregar reinas – lista de n cálculos `addQueens`.
- Correr el Monad combinado y procesar la solución.



Ortogonalidad de los efectos

- Si los efectos de los transformadores son ortogonales, podemos aplicarlos en cualquier orden – conmutan.
- Dos grupos de transformadores
 - **Plomería**: ReaderT, WriterT y StateT.
 - **Control**: ErrorT, MaybeT, ListT y ContT.
- Los transformadores de control no son ortogonales.



Ortogonalidad de los efectos

- En una aplicación que puede fallar, queremos saber las causas.

```
ohnoes :: MonadWriter [String] m => m ()  
ohnoes = do  
  tell ["problem?"]  
  fail "epic"
```



Ortogonalidad de los efectos

- En una aplicación que puede fallar, queremos saber las causas.

```
ohnoes :: MonadWriter [String] m => m ()
ohnoes = do
  tell ["problem?"]
  fail "epic"
```

- ¿Cuál de los siguientes Monad nos ofrece el resultado?

```
type A = WriterT [String] Maybe
type B = MaybeT (Writer [String])

a :: A ()
a = ohnoes

b :: B ()
b = ohnoes
```



Use the types, Luke!

- ¿Cómo usar el monad A? – aplicar el WriterT encima del Maybe

```
> :type runWriterT  
runWriterT :: WriterT w m a -> m (a, w)
```

- Pero m es Maybe.
- Cuando Maybe falla obtenemos Nothing.



Use the types, Luke!

- ¿Cómo usar el monad A? – aplicar el `WriterT` encima del `Maybe`

```
> :type runWriterT
runWriterT :: WriterT w m a -> m (a, w)
```

- Pero `m` es `Maybe`.
 - Cuando `Maybe` falla obtenemos `Nothing`.
- ¿Cómo usar el monad B? – aplicar el `MaybeT` y luego el `Writer`

```
> :type runWriter . runMaybeT
runWriter . runMaybeT :: MaybeT (Writer w) a
                        -> (Maybe a, w)
```

- El resultado está envuelto en `Maybe`.
 - La bitácora es accesible.

¡El orden importa!

```
> runWriterT a
Nothing
> runWriter $ runMaybeT b
(Nothing,["problem?"])
```

- La semántica que buscamos sólo la obtenemos con el Monad B.
- El orden de composición de transformadores se refleja en el orden de composición de sus ejecutores.



Ortogonalidad de los efectos

- ¿Cómo combinar State y Maybe?



Ortogonalidad de los efectos

- ¿Cómo combinar State y Maybe?
- Primera posibilidad

```
type Grok a = MaybeT (State TheState) a
```

- Primero runMaybeT – resulta cómputo de estados.
- Luego aplicar runState – produce un resultado envuelto en Maybe.
- Imperativo – el estado se preserva aún fallando el cómputo.



Ortogonalidad de los efectos

- ¿Cómo combinar State y Maybe?
- Primera posibilidad

```
type Grok a = MaybeT (State TheState) a
```

- Primero runMaybeT – resulta cómputo de estados.
 - Luego aplicar runState – produce un resultado envuelto en Maybe.
 - Imperativo – el estado se preserva aún fallando el cómputo.
- Segunda posibilidad

```
type Grok a = StateT TheState (Maybe a)
```

- Primero runStateT – resulta cómputo de estados envuelto en Maybe.
- Transaccional – transforma el estado sólo si el cómputo tuvo éxito.

¡El orden de los transformadores determina la semántica!



¿Qué es de la vida de Identity?

- Identity actúa como identidad en relación a los transformadores.
- Si MT es el transformador para el Monad M , entonces

$$MT \text{ Identity} == M.$$

- Algunos investigadores sostienen que solamente existen Identity, IO y los transformadores, y que el resto de los Monads se derivan de combinar tantos como se quiera.
 - Puros – terminan en Identity
 - Impuros – terminan en IO



¿Y cómo quedo yo ahí?

- La mayoría de las aplicaciones suele necesitar
 - Configuración – eso es un Reader
 - Estado – eso es un State
 - Bitácora, traza, instrumentación – eso es un Writer
 - Eventualmente, interactuar con el mundo exterior

```
data AppConfig = AppConfig { ... }
data AppState  = AppState  { ... }
data AppLog    = AppLog    { ... }

type MyApp a = WriterT AppLog
                (ReaderT AppConfig
                 (StateT AppState
                  (IO a)))
```



Las librerías estándar

- Paquete `mtl` tradicional de GHC incluye varios módulos – cada uno provee el Monad particular y su transformador.
 - `Control.Monad.Reader`
 - `Control.Monad.Writer`
 - `Control.Monad.State`
 - `Control.Monad.RWS` – ¡Reader + Writer + State!
 - `Control.Monad.List`
 - `Control.Monad.Error`
 - `Control.Monad.Identity`
 - `Control.Monad.Cont` – continuaciones



Las librerías estándar

- Paquete `mtl` tradicional de GHC incluye varios módulos – cada uno provee el Monad particular y su transformador.
 - `Control.Monad.Reader`
 - `Control.Monad.Writer`
 - `Control.Monad.State`
 - `Control.Monad.RWS` – ¡Reader + Writer + State!
 - `Control.Monad.List`
 - `Control.Monad.Error`
 - `Control.Monad.Identity`
 - `Control.Monad.Cont` – continuaciones
- Paquete `transformers` – más moderno y general.
 - `Control.Monad.Trans` – todo está aquí.
 - Provee `MaybeT`.
 - Provee versiones perezosas y ambiciosas para cada Monad.
 - Crea los Monads como Transformadores sobre `Identity`.

Acostúmbrense a usar transformers



Quiero saber más. . .

- All about monads
- transformers vs. mtl
- Documentación de `Control.Monad.State`
- Documentación de `Control.Monad.Reader`
- Documentación de `Control.Monad.Writer`
- Solución monádica para las Ocho Reinas

